Today

- Threads
  - Java API
  - Thread Pools
  - Synchronization

Review

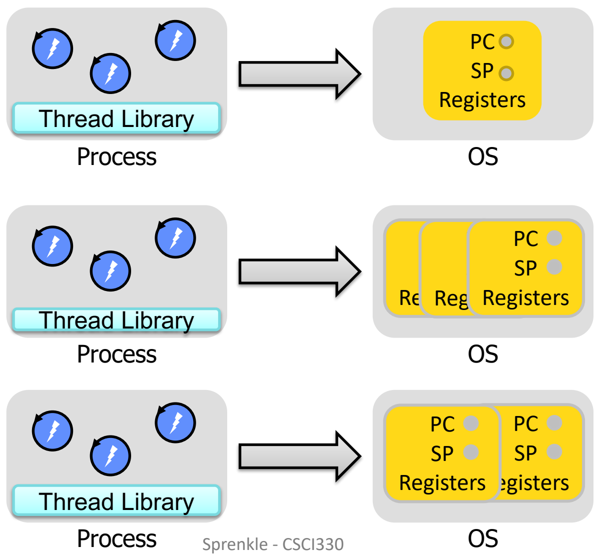
- What is a thread?
  - Why should we consider writing multi-threaded programs?
- What are the 3 main models for implementing threads?
  - What are their tradeoffs?
  - Which model will we assume?
- How do we create threads in Java?

Review: Which threading model would you use in your OS?

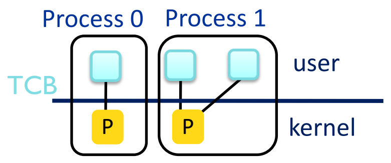
A: (N:1)

B: (1:1)

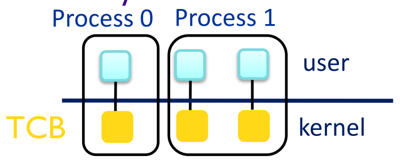
C: (N:M)



Thread Implementation Limitations



- N:1
  - Few systems use because it can't take advantage of multiple processors
- 1:1
  - Creating a user thread requires creating the corresponding kernel thread
    - a large number of kernel threads may burden the performance of a system



- N:M
  - Flexibility → difficult to implement and manage

Why Use Kernel Threads (1:1)?

- I/O: the OS can choose another thread in the same process when a thread does I/O
  - Non-blocking calls are good in theory, but difficult to program in practice
- Simplicity of 1:1 — OS CPU scheduler will do all the scheduling
  - Kernel-level threads can exploit parallelism
  - Different processors of a symmetric multiprocessor
  - Different cores on a multicore CPU
- Used by systems: Linux, Solaris, Windows, pthreads (usually)
- Also used by recent implementations of Java

Review: Java Threads: The Basics

- Extend the Java Thread class

```
class MyThread extends Thread {
    public void run() {
         // do task: your code here
    }
}

…

Thread t1 = new MyThread();
t1.start();
```

Review: Java Threads: The Basics

```
public class RunnableTask implements Runnable {

    public RunnableTask(…) {
        // save any arguments or input for the task
(optional)
    }

    @Override
    public void run() {
        // required to implement for Runnable interface
        …
    }
}
…

RunnableTask task = new RunnableTask();
Thread t1 = new Thread(task, "thread1");
t1.start();
```

Java review:
Tradeoffs of extending vs implementing

Example: Jabber

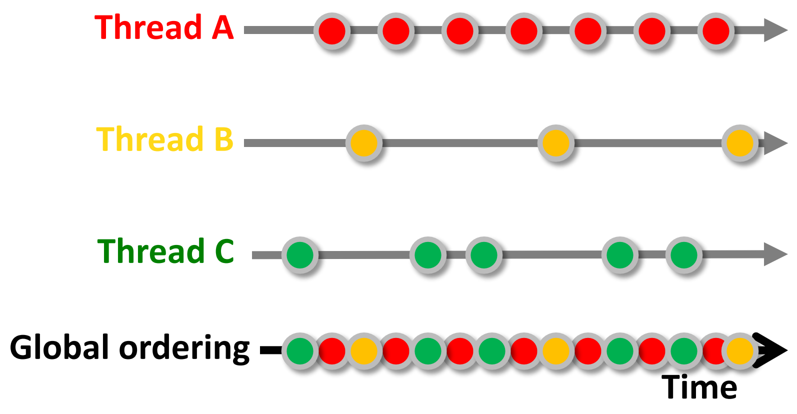
What does this code do?
What will the output be?

```
class Jabber implements Runnable {
        String str;
        public Jabber(String s){ str = s; }
        public void run() {
                while (true) {
                        System.out.print(str);
                        System.out.println();
                }
        }
}
```

```
public class JabberTest {
        public static void main(String[] args) {
                Jabber jabber1 = new Jabber("1");
                Jabber jabber2 = new Jabber("2");
                Thread t1 = new Thread(jabber1);
                Thread t2 = new Thread(jabber2);
                t1.start();
                t2.start();
        }
}
```

JabberTest.java

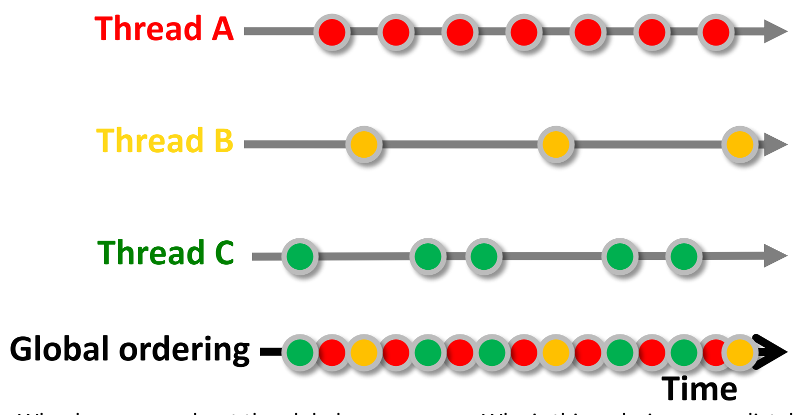
Non-determinism and ordering



- Why do we care about the global ordering?
- Why is this ordering unpredictable?

Oct 17, 2018 Sprenkle - CSCI330 9

Non-determinism and ordering



- Why do we care about the global ordering?
  - Might have dependencies between events
  - Different orderings can produce different results
- Why is this ordering unpredictable?
  - Can't predict how fast processors will run, how threads will be ordered

Oct 17, 2018 Sprenkle - CSCI330 10

Review: Fork Problem

```
Parent's x before wait is 20
Child's x before sleep is 20
Child's x after sleep is 30
Parent's x after wait is 25
Parent's child's status is 0
```

Top two lines of output could be swapped.

```
int main() {
    int x = 20;
    int pid = fork();
    int status;
    if (pid != 0) {
        printf("Parent's x before wait is %d\n",x);
        x = x + 5;
        wait(&status);
        printf("Parent's x after wait is %d\n",x);
        printf("Parent’s child’s status is %d\n",
                    status);
    } else {
        printf("Child's x before sleep is %d\n",x);
        sleep(3);
        x = x + 10;
        printf("Child's x after sleep is %d\n",x);
    }
}
```

Shared Address Space

```
class SASThread extends Thread {
    private int id;
    private int[] array;

    public SASThread(int id, int[] array) {
        this.id = id;
        this.array = array;
    }

    public void run() {
        array[id] = id;
    }
}
```

Shared Address Space

```
public class SharedAddressSpace {
    public static void main(String[] args) {
        int[] vals = {-1, -1, -1};

        Thread t0 = new SASThread(0, vals);
        Thread t1 = new SASThread(1, vals);
        Thread t2 = new SASThread(2, vals);

        t0.start();
        t1.start();
        t2.start();

        try {
            t0.join();
            t1.join();
            t2.join();
        } catch (InterruptedException e) {}

        for( int i=0; i < vals.length; i++ ) {
            System.out.println("vals[" + i + "] = " + vals[i]);
        }
```

THREAD PERFORMANCE

Performance of Thread Maintenance

- Creating a thread is cheaper than creating a new process
  - But it's not free
- Maintaining thread information has an overhead cost
- Common performance issues
  - Creating threads on-the-fly incurs costs
    - increases latency of the task
  - Creating a lot of threads is costly (overhead, maintenance, switching)
    - Recall: throughput graph from scheduling

Review: Throughput: reality

Thrashing, also called congestion collapse
Real servers/devices often have some pathological behaviors at saturation. E.g., they abort requests after investing work in them (thrashing), which wastes work, reducing throughput.

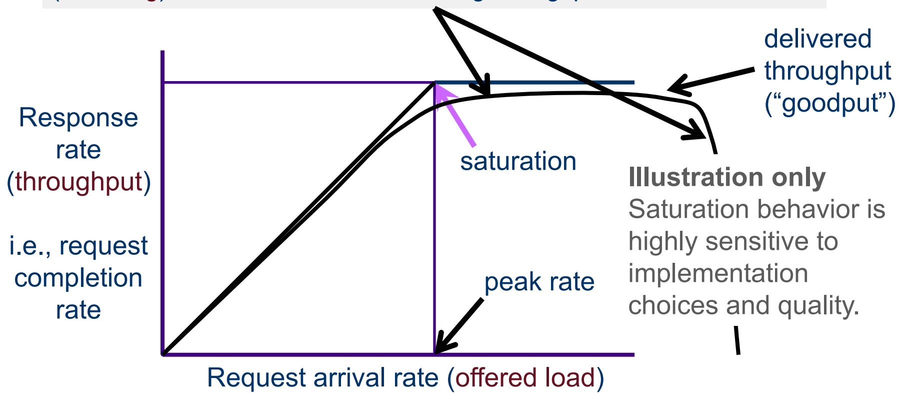
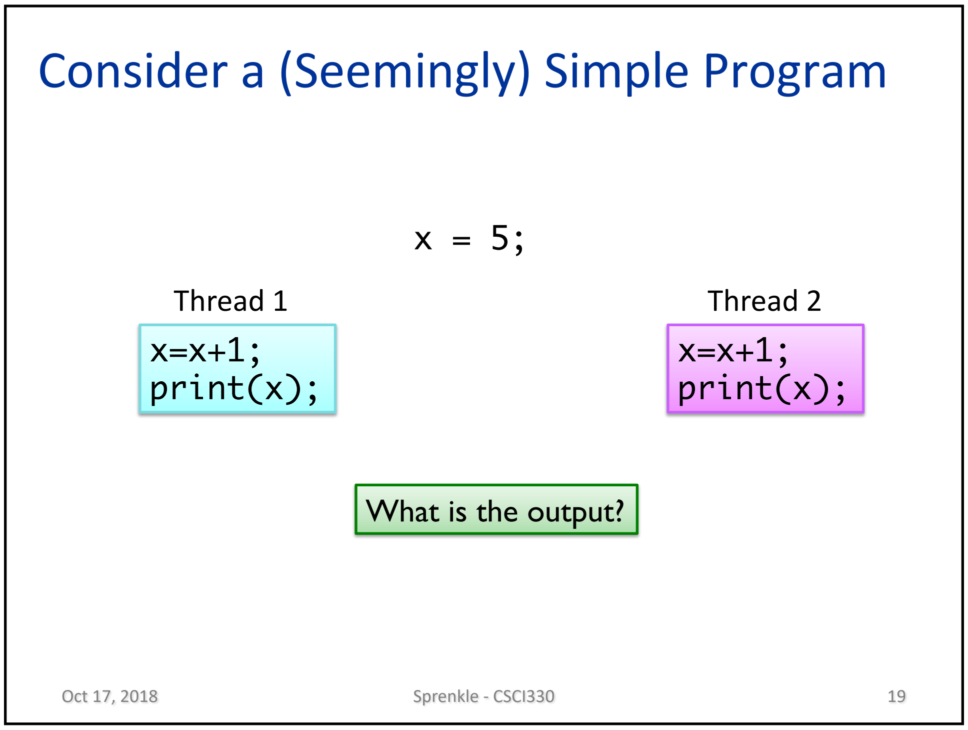


Illustration only
Saturation behavior is highly sensitive to implementation choices and quality.

Solution: Thread Pool

- Create a pool of threads before you need them
  - Incur that cost before work begins
- When a request comes in, assign to an available thread from the pool
  - When thread completes task, return thread to the thread pool
- In Java, *Executors*, *Executor* and *ExecutorService* classes
  - Executors: factory class to create Executor and ExecutorService objects

SYNCHRONIZATION



Consider a (Seemingly) Simple Program

```
x = 5;
```

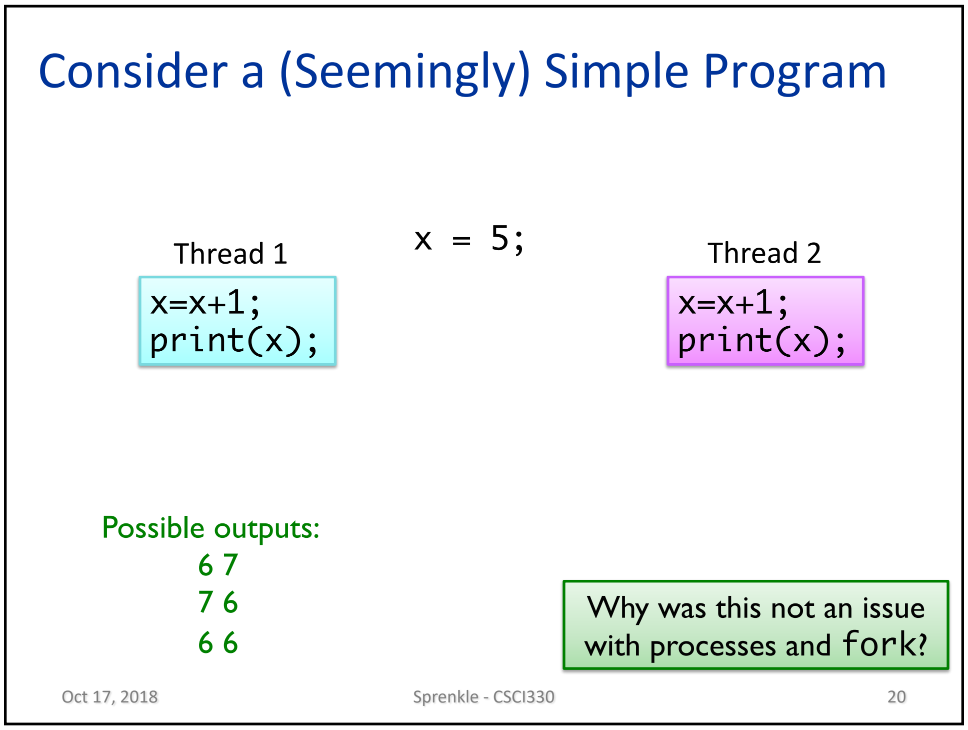
Thread 1

```
x=x+1;
print(x);
```

Thread 2

```
x=x+1;
print(x);
```

What is the output?



Consider a (Seemingly) Simple Program

```
x = 5;
```

Thread 1

```
x=x+1;
print(x);
```

Thread 2

```
x=x+1;
print(x);
```

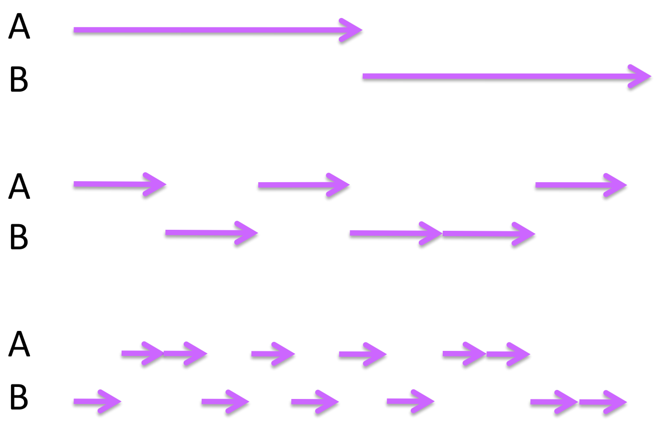
Possible outputs:

```
6 7
7 6
6 6
```

Why was this not an issue with processes and `fork`?

Threads and the Scheduler
(or, Why Multi-threaded Programming is Hard)

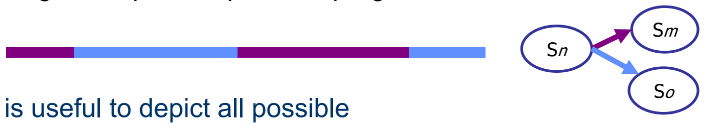
Given two threads, A and B, how might their executions be scheduled?



And, this is with just one processor…

Resource Trajectory Graphs

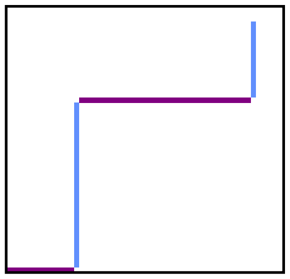
Resource trajectory graphs (RTG) depict the "random walk" through the space of possible program states.



RTG is useful to depict all possible executions of multiple threads.

- I will draw them for only two threads because slides are two-dimensional.
- RTG for N threads is N-dimensional
  - Thread *i* advances along axis *i*.

Each point represents one state in the set of all possible system states.

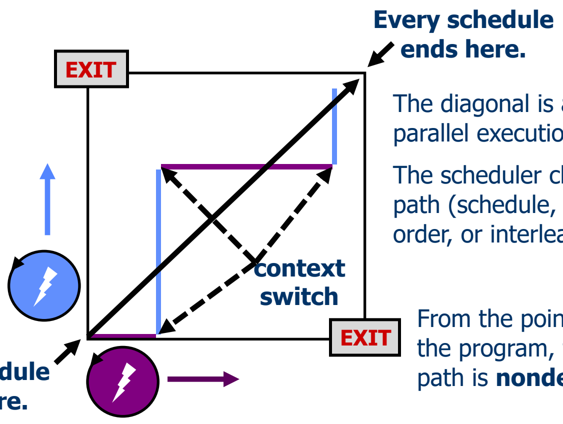


Resource Trajectory Graphs

This RTG depicts a schedule within the space of possible schedules for a simple program of two threads sharing one core.

Blue advances along the y-axis.

Purple advances along the x-axis.

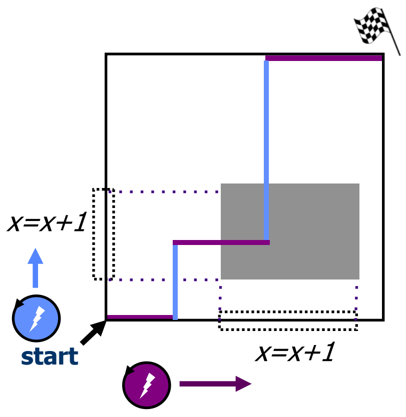


The diagonal is an idealized parallel execution (two cores).

The scheduler chooses the path (schedule, event order, or interleaving).

From the point of view of the program, the chosen path is **nondeterministic**.

A race



This is a valid schedule.

But the schedule interleaves the executions of "x = x + 1" in the two threads.

The variable x is shared.

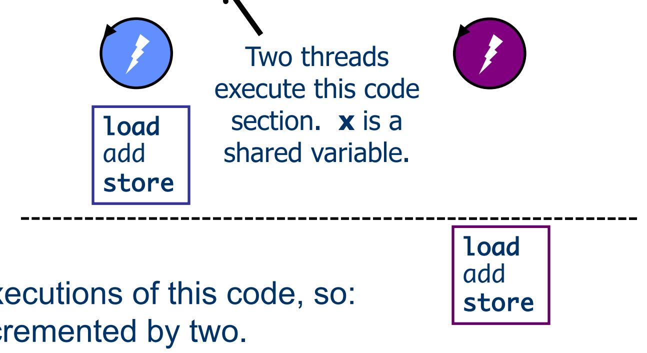
This schedule can corrupt the value of the shared variable x, causing the program to execute incorrectly.

This is an example of a ***race***: the behavior of the program depends on the schedule, and some schedules yield incorrect results.

Reading Between the Lines of C

```
load   x, R2          ; load global variable x
add    R2, 1, R2      ; increment: x = x + 1
store  R2, x          ; store global variable x
```

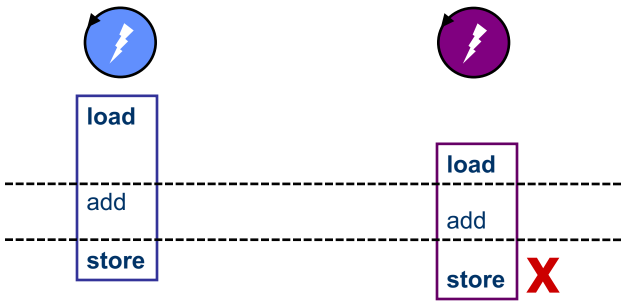
Two threads execute this code section. **x** is a shared variable.



Two executions of this code, so:
x is incremented by two.

Interleaving matters

```
load   x, R2          ; load global variable x
add    R2, 1, R2      ; increment: x = x + 1
store  R2, x          ; store global variable x
```



In this schedule, **x** is incremented only once: last writer wins.
The program breaks under this schedule. This bug is a ***race***.

A ***race condition*** is any situation in which the order of execution affects the final result.

Looking Ahead

- Project 3 – released Friday
  - Big step up on previous projects
- Upcoming Talks
  - Today, Getting a PhD: Myths and Facts, 4 p.m.
    - Parmly 405
  - Monday, 12:15 p.m. Alicia Bargar
    - Sci Addn 202A – pizza lunch!

Exam Logistics

- 2 hours to take exam
- Open notes, my slides, textbook
- Closed [other] internet
  - Do not download Wikipedia pages or do other things that you're pretty sure I wouldn't want you to do
- 3 Sections
  - Very Short Answer
  - Short Answer
  - Applied

Recommendations

- Write your answers in Word and then copy to Sakai to avoid issues with the site timing out
- Answer what the question asks and only that
  - Assemble thoughts, then write
- Be cognizant of the question's point values
  - Ex: Don't spend a ton of time on a 4-point question
- If you need to look up the answer, skip the question and come back to it later
  - Overhead costs of searching